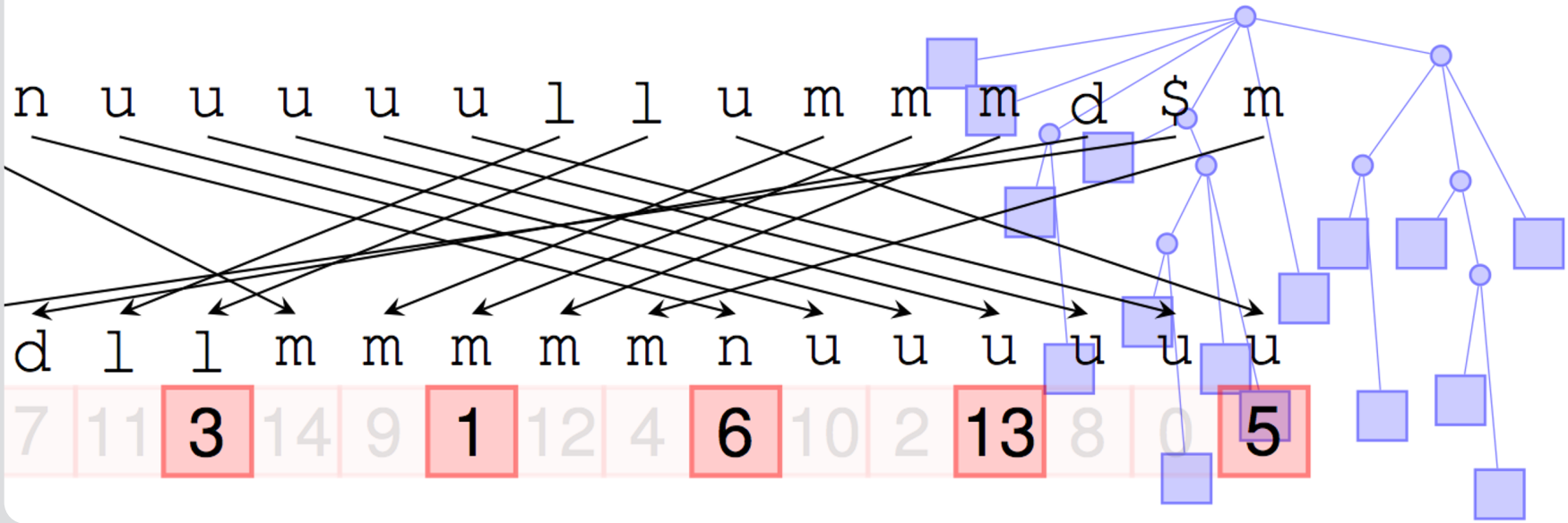
Text Indexing: Lecture 1

Simon Gog – *gog@kit.edu*

Institute of Theoretical Informatics - Algorithmics



Notation

String

- A *string* S is a sequence of *symbols* or *characters*.
- Usually we use $n = |S|$ to denote its length.
- Each character is an element of a finite ordered set $\Sigma = [0..\sigma - 1]$, called the alphabet.
- The alphabet size is denoted as $\sigma = |\Sigma|$.
- The i-th character of S is $S[i]$.
- The *substring* from position i to j is $S[i..j]$.
- A substring with $i = 0$ is called *prefix*.
- A substring with $j = n - 1$ is called *suffix*.
- The i-th *suffix* is $S[i..n - 1]$.

Notation

- Given strings S, S'. Appending the characters of S and S' results in the *concatenation* SS'.
- Concatenation is also defined for single character strings c.
- *Lexicographical order* „$<$”. Let a, b be characters and X, Y be strings. Then $aX < bY$, if $a < b$ or $a = b$ and $X < Y$.

Classical Text Indexes

Tries or Digital Trees

- First structures for string search
- References: de la Briadnais (1959), Fredikin 1960

Definition

A *trie* for a set S of distinct strings $S_0, S_1, \ldots, S_{N-1}$ is a tree where each node represents a distinct prefix in the set. The root node represents the empty prefix ϵ. Node v representing prefix Y is a child of node u representing prefix X iff $Y = Xc$ for some character c, which will label the tree edge from u to v.

If all S_i end with a special $\$$ sentinel (which does not occur elsewhere in the text), no S_i is a prefix of another string. $\Rightarrow$ trie has N leaves.

Classical Text Indexes

Tries or Digital Trees

Time complexities

- Construction: $\mathcal{O}(|S_0| + \ldots + |S_{N-1}|)$
- Search:
  - $\mathcal{O}(m\sigma)$ – list
  - $\mathcal{O}(m \log \sigma)$ – array
  - $\mathcal{O}(m)$ – hash table

Classical Text Indexes

Suffix Tries and Suffix Trees

Suffix Trie

The *suffix trie* of a text $\mathcal{T}$ is a trie data structure built over all suffixes of $\mathcal{T}$.

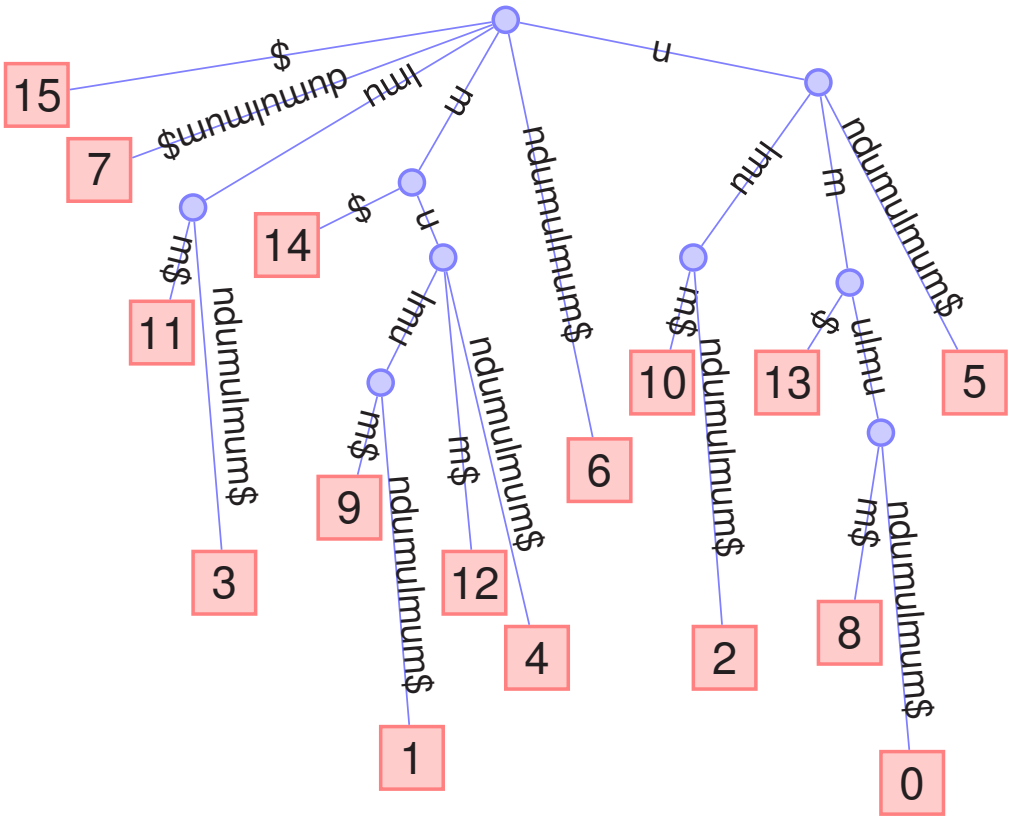
- Space: $\mathcal{O}(n^2)$ words or $\mathcal{O}(n^2 \log \sigma + n^2 \log n)$ bits

Suffix Tree

The *suffix tree* of a text $\mathcal{T}$ is a suffix trie where each unary path is converted into a single edge. Those edges are labeled by strings obtained by concatenating the characters of the replaced path. The leaves of the suffix tree indicate the text position where the corresponding suffixes start.

Classical Text Indexes

Suffix Trees



- $\mathcal{O}(n)$ construction (Weiner 1973, McCreight 1976, Ukkonen 1995)
- $\mathcal{O}(n \log n + n \log \sigma)$ space
  - Labels not stored explicitly (but pointer to $\mathcal{T}$)
  - Length of labels/depth of nodes stored
- Problem: Index size much larger than original text. $> 20x$ text size (dependent on functionality)

Suffix Array

i	$SA[i]$	$\mathcal{T}[SA[i]..n-1]$
0	18	\$
1	17	a\$
2	10	abarbara\$
3	7	abrabarbara\$
4	0	abracadabrabarbara\$
5	3	acadabrabarbara\$
6	5	adabrabarbara\$
7	15	ara\$
8	12	arbara\$
9	14	bara\$
10	11	barbara\$
11	8	brabarbara\$
12	1	bracadabrabarbara\$
13	4	cadabrabarbara\$
14	6	dabrabarbara\$
15	16	ra\$
16	9	rabarbara\$
17	2	racadabrabarbara\$
18	13	rbara\$

- $\mathcal{T}[SA[i]..n-1] < \mathcal{T}[SA[i+1]..n-1]$
- $SA[i]$ contains the starting position of the ith lex. smallest suffix of $\mathcal{T}$.
- Matching algorithm: binary search (forward, left-to-right)

FM-Index

(Ferragina &Manzini [1])

- Index based on Burrows-Wheeler-Transform (BWT)
- Matching algorithm works backwards (right-to-left)
- Existence and count queries in time $\mathcal{O}(m \log \sigma)$

BWT

- $BWT[i] = \mathcal{T}[SA[i] - 1 \bmod n]$
- uncompressed size: $n \log \sigma$ bits
- compressed size: $nH_k(\mathcal{T})$ bits (+information for contexts of length k)

Backward Search

i	$SA[i]$	BWT	$\mathcal{T}[SA[i]..n-1]$
0	18	a	$
1	17	r	a$
2	10	r	abarbara$
3	7	d	abrabarbara$
4	0	$	abracadabrabarbara$
5	3	r	acadabrabarbara$
6	5	c	adabrabarbara$
7	15	b	ara$
8	12	b	arbara$
9	14	r	bara$
10	11	a	barbara$
11	8	a	brabarbara$
12	1	a	bracadabrabarbara$
13	4	a	cadabrabarbara$
14	6	a	dabrabarbara$
15	16	a	ra$
16	9	b	rabarbara$
17	2	b	racadabrabarbara$
18	13	a	rbara$

- $BWT[i] = \mathcal{T}[SA[i]-1]$, for $SA[i] > 0$
- $BWT[i] = \mathcal{T}[n-1]$, for $SA[i] = 0$
- I.e. $BWT[i]$ is the character preceding suffix $SA[i]$

Backward Search

i	BWT	$\mathcal{T}[SA[i]..n-1]$
0	a	$
1	r	a$
2	r	abarbara$
3	d	abrabarbara$
4	$	abracadabrabarbara$
5	r	acadabrabarbara$
6	c	adabrabarbara$
7	b	ara$
8	b	arbara$
9	r	bara$
10	a	barbara$
11	a	brabarbara$
12	a	bracadabrabarbara$
13	a	cadabrabarbara$
14	a	dabrabarbara$
15	a	ra$
16	b	rabarbara$
17	b	racadabrabarbara$
18	a	rbara$

Array C contains for each $c \in \Sigma$ the position of the first suffix in SA which starts with c:

$	a	b	c	d	r	r+1
0	1	9	13	14	15	19

- Operation $rank(i, X, BWT)$ returns how often character $X \in \Sigma$ occurs in the prefix $BWT[0..i-1]$.
- Example: search for $\mathcal{P} = bar$.

Backward Search

i	BWT	$\mathcal{T}[SA[i]..n-1]$
0	a	$
1	r	a$
2	r	abarbara$
3	d	abrabarbara$
4	$	abracadabrabarbara$
5	r	acadabrabarbara$
6	c	adabrabarbara$
7	b	ara$
8	b	arbara$
9	r	bara$
10	a	barbara$
11	a	brabarbara$
12	a	bracadabrabarbara$
13	a	cadabrabarbara$
14	a	dabrabarbara$
15	a	ra$
16	b	rabarbara$
17	b	racadabrabarbara$
18	a	rbara$

C

$	a	b	c	d	r
0	1	9	13	14	15

- Search backwards for *bar*.
- Initial interval: $[sp_0, ep_0] = [0..n-1]$
- Determine interval for r:
  $sp_1 = C[r] + rank(sp_0, r, BWT)$
  $ep_1 = C[r] + rank(ep_0 + 1, r, BWT) - 1$

Backward Search

i	BWT	$\mathcal{T}[SA[i]..n-1]$
0	a	$
1	r	a$
2	r	abarbara$
3	d	abrabarbara$
4	$	abracadabrabarbara$
5	r	acadabrabarbara$
6	c	adabrabarbara$
7	b	ara$
8	b	arbara$
9	r	bara$
10	a	barbara$
11	a	brabarbara$
12	a	bracadabrabarbara$
13	a	cadabrabarbara$
14	a	dabrabarbara$
15	a	ra$
16	b	rabarbara$
17	b	racadabrabarbara$
18	a	rbara$

C					
$	a	b	c	d	r
0	1	9	13	14	15

- Search backwards for *bar*.
- Initial interval: $[sp_0, ep_0] = [0..n-1]$
- Determine interval for r:
  $sp_1 = 15 + rank(0, r, BWT)$
  $ep_1 = 15 + rank(19, r, BWT)$

Backward Search

i	BWT	$\mathcal{T}[SA[i]..n-1]$
0	a	$
1	r	a$
2	r	abarbara$
3	d	abrabarbara$
4	$	abracadabrabarbara$
5	r	acadabrabarbara$
6	c	adabrabarbara$
7	b	ara$
8	b	arbara$
9	r	bara$
10	a	barbara$
11	a	brabarbara$
12	a	bracadabrabarbara$
13	a	cadabrabarbara$
14	a	dabrabarbara$
15	a	ra$
16	b	rabarbara$
17	b	racadabrabarbara$
18	a	rbara$

	C				
$	a	b	c	d	r
0	1	9	13	14	15

- Search backwards for *bar*.
- Initial interval: $[sp_0, ep_0] = [0..n-1]$
- Determine interval for r:
  $sp_1 = 15 + 0$
  $ep_1 = 15 + rank(19, r, BWT) - 1$

Backward Search

i	BWT	$\mathcal{T}[SA[i]..n-1]$
0	a	$
1	r	a$
2	r	abarbara$
3	d	abrabarbara$
4	$	abracadabrabarbara$
5	r	acadabrabarbara$
6	c	adabrabarbara$
7	b	ara$
8	b	arbara$
9	r	bara$
10	a	barbara$
11	a	brabarbara$
12	a	bracadabrabarbara$
13	a	cadabrabarbara$
14	a	dabrabarbara$
15	a	ra$
16	b	rabarbara$
17	b	racadabrabarbara$
18	a	rbara$

C					
$	a	b	c	d	r
0	1	9	13	14	15

- Search backwards for *bar*.
- Initial interval: $[sp_0, ep_0] = [0..n-1]$
- Determine interval for r:
  $sp_1 = 15+0 = 15$
  $ep_1 = 15+4-1 = 18$

Backward Search

i	BWT	$\mathcal{T}[SA[i]..n-1]$
0	a	$
1	r	a$
2	r	abarbara$
3	d	abrabarbara$
4	$	abracadabrabarbara$
5	r	acadabrabarbara$
6	c	adabrabarbara$
7	b	ara$
8	b	arbara$
9	r	bara$
10	a	barbara$
11	a	brabarbara$
12	a	bracadabrabarbara$
13	a	cadabrabarbara$
14	a	dabrabarbara$
15	a	ra$
16	b	rabarbara$
17	b	racadabrabarbara$
18	a	rbara$

C					
$	a	b	c	d	r
0	1	9	13	14	15

- Search backwards for *bar*.
- Interval: $[sp_1, ep_1] = [15..18]$
- Determine interval for *ar*:
  $sp_2 = C[a] + rank(sp_1 - 1, a, BWT)$
  $ep_2 = C[a] + rank(ep_1, a, BWT) - 1$

Backward Search

i	BWT	$\mathcal{T}[SA[i]..n-1]$
0	a	$
1	r	a$
2	r	abarbara$
3	d	abrabarbara$
4	$	abracadabrabarbara$
5	r	acadabrabarbara$
6	c	adabrabarbara$
7	b	ara$
8	b	arbara$
9	r	bara$
10	a	barbara$
11	a	brabarbara$
12	a	bracadabrabarbara$
13	a	cadabrabarbara$
14	a	dabrabarbara$
15	a	ra$
16	b	rabarbara$
17	b	racadabrabarbara$
18	a	rbara$

C					
$	a	b	c	d	r
0	1	9	13	14	15

- Search backwards for *bar*.
- Interval: $[sp_1, ep_1] = [15..18]$
- Determine interval for *ar*:
  $sp_2 = 1 + rank(15, a, BWT)$
  $ep_2 = 1 + rank(ep_1, a, BWT)$

Backward Search

i	BWT	$\mathcal{T}[SA[i]..n-1]$
0	a	$
1	r	a$
2	r	abarbara$
3	d	abrabarbara$
4	$	abracadabrabarbara$
5	r	acadabrabarbara$
6	c	adabrabarbara$
7	b	ara$
8	b	arbara$
9	r	bara$
10	a	barbara$
11	a	brabarbara$
12	a	bracadabrabarbara$
13	a	cadabrabarbara$
14	a	dabrabarbara$
15	a	ra$
16	b	rabarbara$
17	b	racadabrabarbara$
18	a	rbara$

C					
$	a	b	c	d	r
0	1	9	13	14	15

- Search backwards for *bar*.
- Interval: $[sp_1, ep_1] = [15..18]$
- Determine interval for *ar*:
  $sp_2 = 1 + rank(15, a, BWT)$
  $ep_2 = 1 + rank(ep_1, a, BWT)$

Backward Search

i	BWT	$\mathcal{T}[SA[i]..n-1]$
0	a	$
1	r	a$
2	r	abarbara$
3	d	abrabarbara$
4	$	abracadabrabarbara$
5	r	acadabrabarbara$
6	c	adabrabarbara$
7	b	ara$
8	b	arbara$
9	r	bara$
10	a	barbara$
11	a	brabarbara$
12	a	bracadabrabarbara$
13	a	cadabrabarbara$
14	a	dabrabarbara$
15	a	ra$
16	b	rabarbara$
17	b	racadabrabarbara$
18	a	rbara$

C					
$	a	b	c	d	r
0	1	9	13	14	15

- Search backwards for *bar*.
- Interval: $[sp_1, ep_1] = [15..18]$
- Determine interval for *ar*:
  $sp_2 = 1 + 6$
  $ep_2 = 1 + rank(19, a, BWT) - 1$

Backward Search

i	BWT	$\mathcal{T}[SA[i]..n-1]$
0	a	$
1	r	a$
2	r	abarbara$
3	d	abrabarbara$
4	$	abracadabrabarbara$
5	r	acadabrabarbara$
6	c	adabrabarbara$
7	b	ara$
8	b	arbara$
9	r	bara$
10	a	barbara$
11	a	brabarbara$
12	a	bracadabrabarbara$
13	a	cadabrabarbara$
14	a	dabrabarbara$
15	a	ra$
16	b	rabarbara$
17	b	racadabrabarbara$
18	a	rbara$

C					
$	a	b	c	d	r
0	1	9	13	14	15

- Search backwards for *bar*.
- Interval: $[sp_1, ep_1] = [15..18]$
- Determine interval for *ar*:
  $sp_2 = 1+6 = 7$
  $ep_2 = 1+8-1 = 8$

Backward Search

i	BWT	$\mathcal{T}[SA[i]..n-1]$
0	a	$
1	r	a$
2	r	abarbara$
3	d	abrabarbara$
4	$	abracadabrabarbara$
5	r	acadabrabarbara$
6	c	adabrabarbara$
7	b	ara$
8	b	arbara$
9	r	bara$
10	a	barbara$
11	a	brabarbara$
12	a	bracadabrabarbara$
13	a	cadabrabarbara$
14	a	dabrabarbara$
15	a	ra$
16	b	rabarbara$
17	b	racadabrabarbara$
18	a	rbara$

C					
$	a	b	c	d	r
0	1	9	13	14	15

- Search backwards for *bar*.
- Interval: $[sp_2, ep_2] = [7..8]$
- Determine interval for *bar*:
  $sp_3 = C[b] + rank(sp_2 - 1, b, BWT)$
  $ep_3 = C[b] + rank(ep_2, b, BWT) - 1$

Backward Search

i	BWT	$\mathcal{T}[SA[i]..n-1]$
0	a	$
1	r	a$
2	r	abarbara$
3	d	abrabarbara$
4	$	abracadabrabarbara$
5	r	acadabrabarbara$
6	c	adabrabarbara$
7	b	ara$
8	b	arbara$
9	r	bara$
10	a	barbara$
11	a	brabarbara$
12	a	bracadabrabarbara$
13	a	cadabrabarbara$
14	a	dabrabarbara$
15	a	ra$
16	b	rabarbara$
17	b	racadabrabarbara$
18	a	rbara$

C

$	a	b	c	d	r
0	1	9	13	14	15

- Search backwards for *bar*.
- Interval: $[sp_2, ep_2] = [7..8]$
- Determine interval for *bar*:
  $sp_3 = 9 + rank(7, b, BWT)$
  $ep_3 = 9 + rank(ep_1, b, BWT)$

Backward Search

i	BWT	$\mathcal{T}[SA[i]..n-1]$
0	a	$
1	r	a$
2	r	abarbara$
3	d	abrabarbara$
4	$	abracadabrabarbara$
5	r	acadabrabarbara$
6	c	adabrabarbara$
7	b	ara$
8	b	arbara$
9	r	bara$
10	a	barbara$
11	a	brabarbara$
12	a	bracadabrabarbara$
13	a	cadabrabarbara$
14	a	dabrabarbara$
15	a	ra$
16	b	rabarbara$
17	b	racadabrabarbara$
18	a	rbara$

C					
$	a	b	c	d	r
0	1	9	13	14	15

- Search backwards for *bar*.
- Interval: $[sp_2, ep_2] = [7..8]$
- Determine interval for *bar*:
  $sp_3 = 9 + rank(7, b, BWT)$
  $ep_3 = 9 + rank(ep_1, b, BWT)$

Backward Search

i	BWT	$\mathcal{T}[SA[i]..n-1]$
0	a	$
1	r	a$
2	r	abarbara$
3	d	abrabarbara$
4	$	abracadabrabarbara$
5	r	acadabrabarbara$
6	c	adabrabarbara$
7	b	ara$
8	b	arbara$
9	r	bara$
10	a	barbara$
11	a	brabarbara$
12	a	bracadabrabarbara$
13	a	cadabrabarbara$
14	a	dabrabarbara$
15	a	ra$
16	b	rabarbara$
17	b	racadabrabarbara$
18	a	rbara$

C					
$	a	b	c	d	r
0	1	9	13	14	15

- Search backwards for *bar*.
- Interval: $[sp_2, ep_2] = [7..8]$
- Determine interval for *bar*:
  $sp_3 = 9+0$
  $ep_3 = 9+rank(9, b, BWT)-1$

Backward Search

i	BWT	$\mathcal{T}[SA[i]..n-1]$
0	a	$
1	r	a$
2	r	abarbara$
3	d	abrabarbara$
4	$	abracadabrabarbara$
5	r	acadabrabarbara$
6	c	adabrabarbara$
7	b	ara$
8	b	arbara$
9	r	bara$
10	a	barbara$
11	a	brabarbara$
12	a	bracadabrabarbara$
13	a	cadabrabarbara$
14	a	dabrabarbara$
15	a	ra$
16	b	rabarbara$
17	b	racadabrabarbara$
18	a	rbara$

C					
$	a	b	c	d	r
0	1	9	13	14	15

- Search backwards for *bar*.
- Interval: $[sp_2, ep_2] = [7..8]$
- Determine interval for *bar*:
  $sp_3 = 9+0 = 9$
  $ep_3 = 9+2-1 = 10$

Backward Search

Summary

- Only C and a data structure R supporting the *rank* operation on *BWT* are required for existence and count queries.
- Space: $\sigma \log n$ bits for C + space for R
- Time: $\mathcal{O}(m \cdot t_{rank})$, where t_{rank} is time for one rank operation. Independent from n?
- Next: How to implement *rank*?

Rank operation

- Constant time and $o(n)$ extra space solution on bitvectors (Jacobson [3])
- Solution on general sequences: Wavelet Tree (Grossi & Vitter [2])

Backward Search

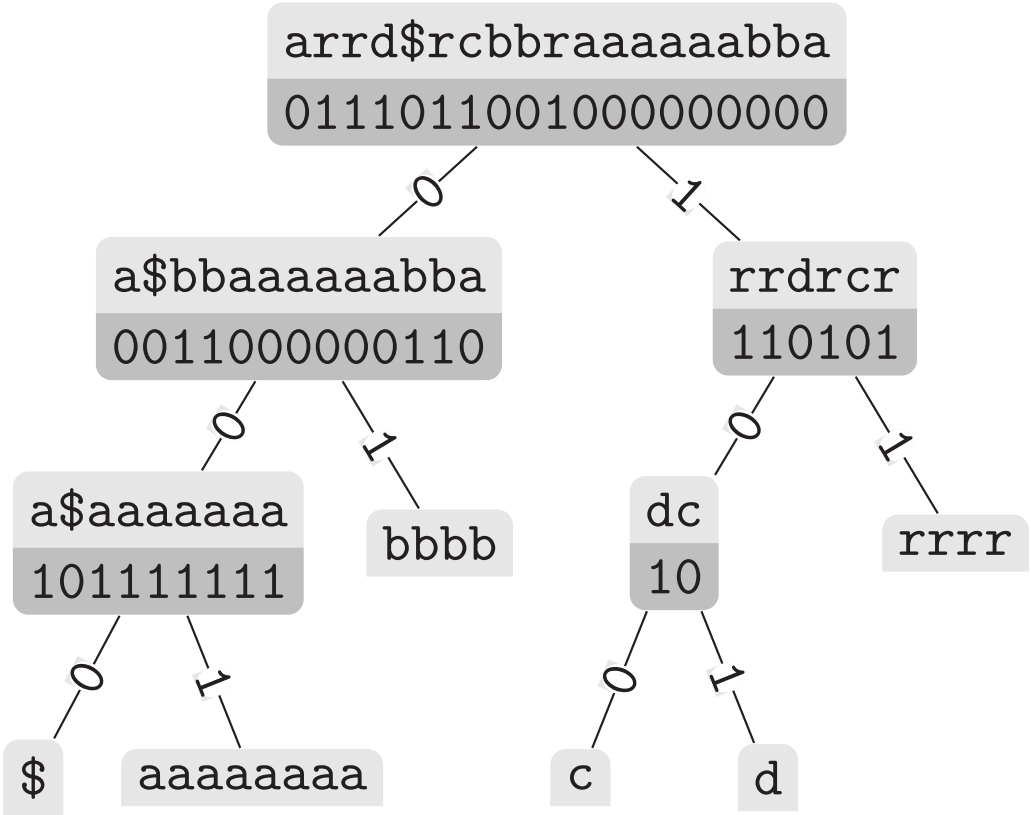
Summary

- Only C and a data structure R supporting the *rank* operation on *BWT* are required for existence and count queries.
- Space: $\sigma \log n$ bits for C + space for R
- Time: $\mathcal{O}(m \cdot t_{rank})$, where t_{rank} is time for one rank operation. Independent from n? If t_{rank} is independent from n
- Next: How to implement *rank*?

Rank operation

- Constant time and $o(n)$ extra space solution on bitvectors (Jacobson [3])
- Solution on general sequences: Wavelet Tree (Grossi & Vitter [2])

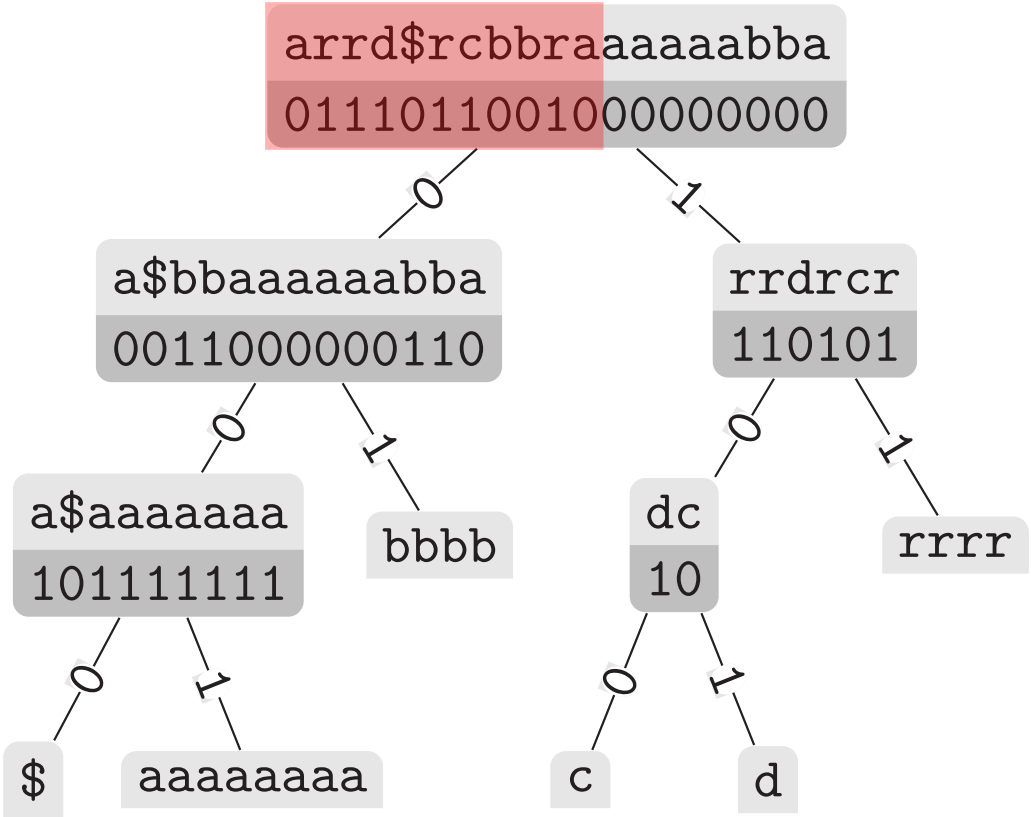
Wavelet Tree Example: Calculate Rank



a = 001

$$rank(11, a, WT) = rank(rank(rank(11, 0, b_\epsilon) = 5, 0, b_0) = 3, 1, b_{00}) = 2$$

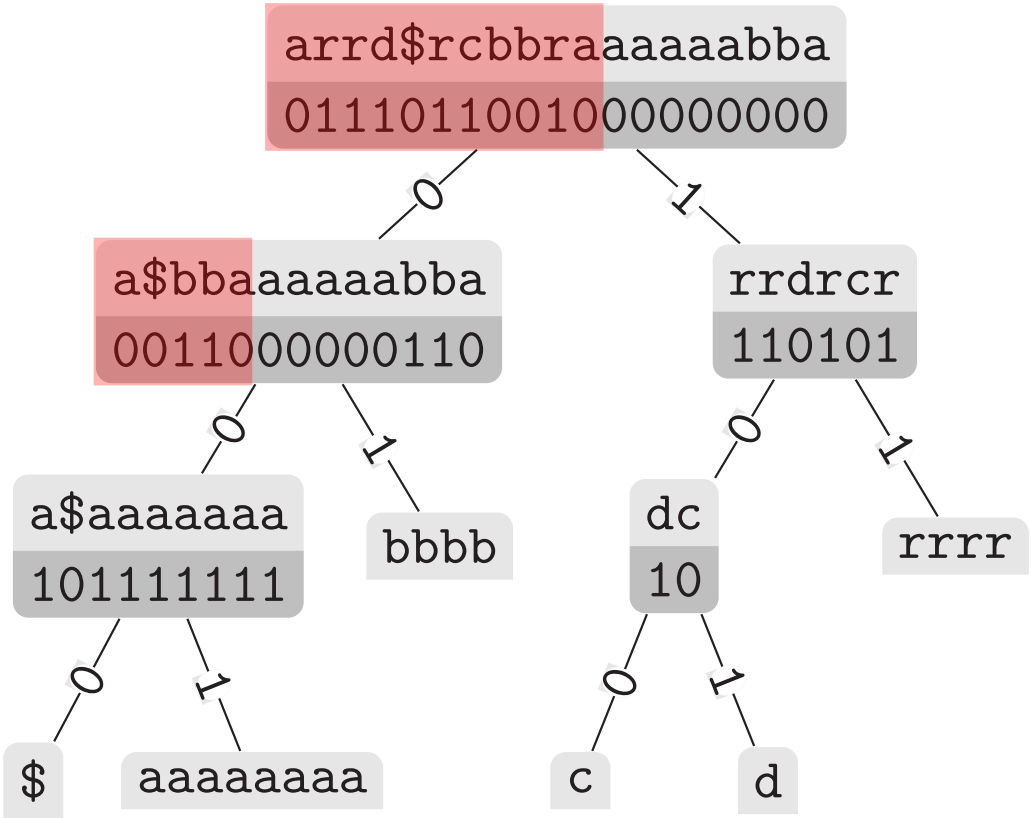
Wavelet Tree Example: Calculate Rank



a = 001

$$rank(11, a, WT) = rank(rank(rank(11, 0, b_\epsilon) = 5, 0, b_0) = 3, 1, b_{00}) = 2$$

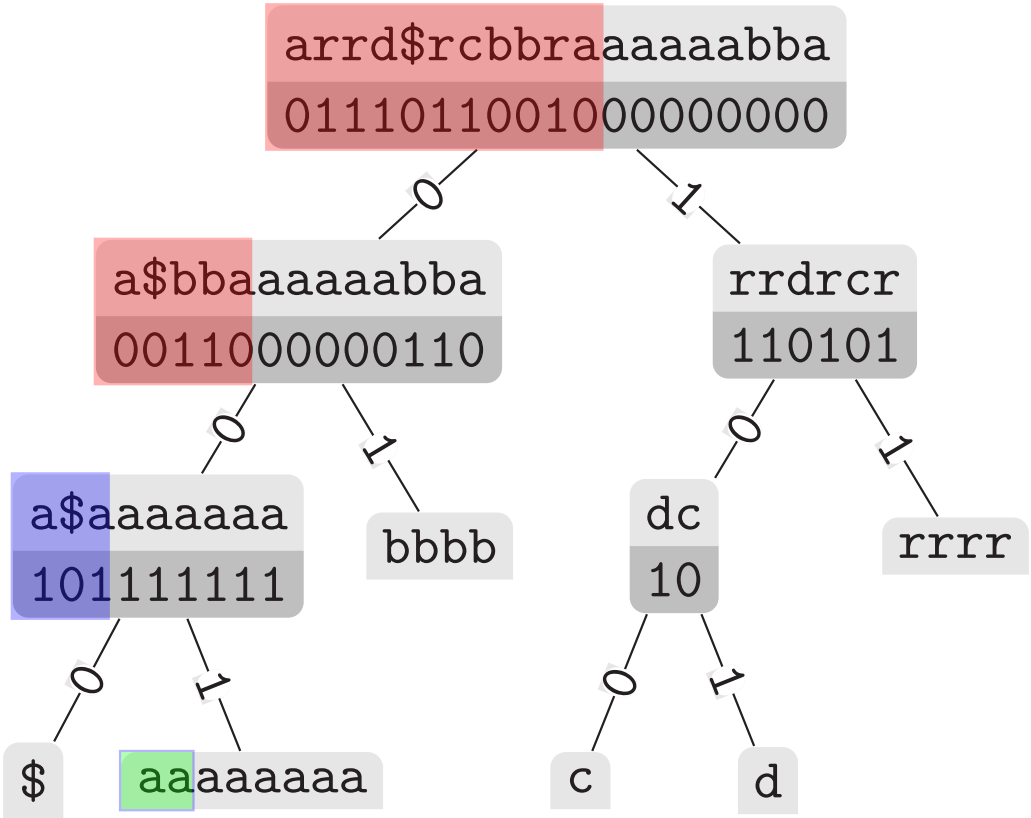
Wavelet Tree Example: Calculate Rank



a = 001

$$rank(11, a, WT) = rank(rank(rank(11, 0, b_\epsilon) = 5, 0, b_0) = 3, 1, b_{00}) = 2$$

Wavelet Tree Example: Calculate Rank



a = 001

$$rank(11, a, WT) = rank(rank(rank(11, 0, b_\epsilon) = 5, 0, b_0) = 3, 1, b_{00}) = 2$$

$o(n)$ space / constant query time

- Given bitvector B of length n bits
- Divide B into superblocks of size L
- For each superblock SB_j store $\Sigma_{i=0}^{(j-1)L-1} B[i]$ in $\log n$ bits
- Divide each superblock into blocks of size S
- For each block B_k of superblock j store $\Sigma_{i=(j-1)L}^{(j-1)L+kS-1} B[i]$ in $\log L$ bits

If blocks are small enough, we can pre-compute a table which stores all answers for every block and position (four Russian-Tick).

Solution

- $L = \log^2 n$
- $S = \frac{1}{2} \log n$

Final space: $\mathcal{O}(\frac{n}{\log n} + \frac{n \log\log n}{\log n} + \sqrt{n} \log n \log\log n)$ bits

Turning the FM-Index into a Self Index

Self Index

Does not only provide search functionality but also efficient reconstruction of any substring of the original text.

LF mapping

For every suffix $j = SA[i]$, $LF(i)$ is the position of $j - 1$ (the previous suffix in the text) in SA. It holds:

$$LF[i] = C[BWT[i]] + rank(i, BWT[i], BWT)$$

I.e. we can decode text backwards. Starting from the last suffix $ at position 0, we can decode the whole text.

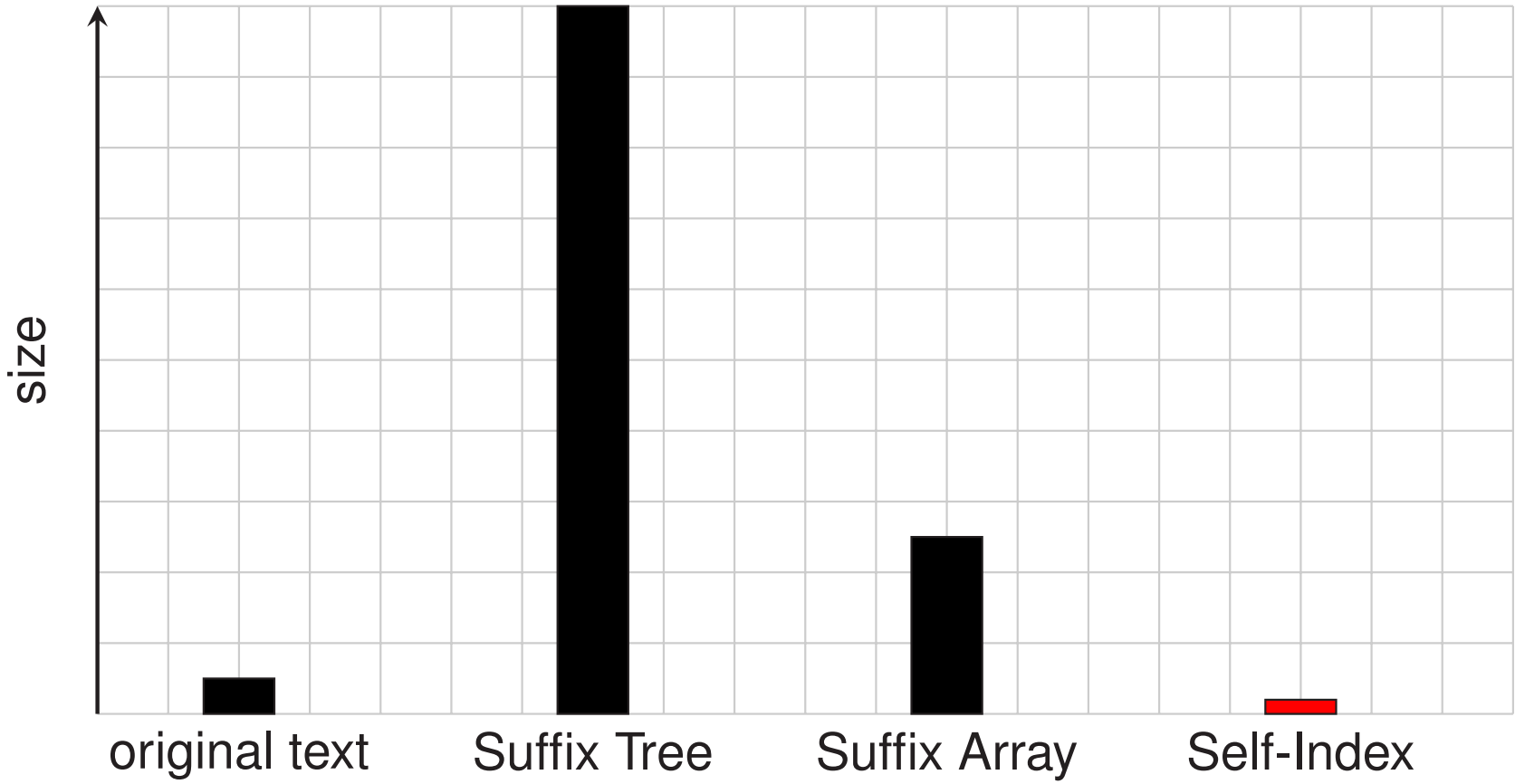
Turning the FM-Index into a Self-Index

Sampling (for locate)

Fix a sampling rate s. Add a bitvector B of length n with $B[i] = 1$ if $SA[i] \equiv 0 \mod s$. Store the samples in array SA' of size n/s.
I.e. for all i with $B[i] = 1$, $SA'[rank(i, 1, B)] = SA[i]$.

Pseudo-code for accessing SA[i]

Index size comparison (English text)



Implementation (in SDSL)

- Plain bitvector: `bit_vector`
- Rank support: `rank_support_v<>` (proposed by S. Vigna, [4])
- Balanced wavelet tree: `wt_blcd<>`
- FM-Index: `csa_wt<>`

Bibliography

[1] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Proc. FOCS*, pages 390–398, 2000.

[2] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proc. SODA*, pages 841–850, 2003.

[3] Guy Jacobson. Space-efficient static trees and graphs. In *Proc. FOCS*, pages 549–554, 1989.

[4] Sebastiano Vigna. Broadword implementation of rank/select queries. In *Proc. WEA*, pages 154–168, 2008.